Query System & Tag Transfer Details

Eric Liang

April 15, 2012

This rough guide is intended to supplement the documentation in the source code and the README.txt file.

1 Control Flow

The entry point to running a query is [querySystem.py] or [querySystemCopy.py] (there are two just for the sake of convenience). This imports the Context object, sets the parameters, and runs the query using [system.py].

- [system.py] can be thought of as collection of functions that operate on individual queries (and the characterize() function calls these in the right order).
- characterize() calls match(C, Q) for each individual query image.
- match() runs a query [query.py] and gets back a list of feature matches.
- These feature matches are put through a series of strong filtering stages to discard bad feature matches, and are reranked.
- Then the top image is picked. This would be the "topN" result.
- Following that, homographies are computed between the query and database matches. This process continues down the topN list until a "good" homography match is determined. Then tags are transferred onto this image and returned to the client.

2 Configuration

[config.py] contains some deprecated options, you need not worry about them. However, the INFO(x) and INFO_TIMING(x) functions provide an easy way to output debug messages. They print time and thread information.

When running the query system, the following environment variables may apply:

- DEBUG: if DEBUG is set in the environment, only a single process will be used to do homography calcuations/draw images (to avoid intermingling of messages and lost exceptions). Note that to enable multiprocessing wrapping the characterize statement with a "with" statement:

```
with system.Multiprocessing():
  system.characterize(C)
```

- Note that this is not related to the number of threads used when running database queries, which is managed by estimate_threads_avail() in [system.py].
- NO_HOM: if this is set the the query system will only compute the topN images matches and will not proceed with further processing.
- NO_DRAW: if this is set the query system will do homography/pose estimation logic, but will not actually draw an output image.

2.1 The Context and Query classes [context.py]

2.1.1 Query

In general, Query objects are named Q and contain gps location, sensor data, image path, etc. Context objects C contain configuration information about what database we are using, the set of Q objects to iterate over, etc.

The Query object contains the pgm_scale parameter, which is used by the tag drawing code to determine how to scale the coordinate system to the output image. It should be the ratio of the size of .pgm file the sift features were extracted from and the jpg image the tags are to be drawn onto. This parameter is set by the Context object in a few of the query sets already, so if you aren't sure what it should be look at those examples.

- Note that individual database images have no representation besides their sift feature-file name. (This is probably unfortunate). Most commonly we refer to them as "siftname" or "matchedimg". These are usually of the form [lat],[lon],[view].sift.txt

2.1.2 Context

The Context class "holds" run-specific configuration information that is passed to the query system. To construct a context object, do

```
from context import DEFAULT_CONTEXT
C = DEFAULT_CONTEXT.copy()
C.query = "query4"
C.params = {
  'dist_threshold': 70000
}
# etc.
```

See [querySystem.py] for an example of this, and [context.py] for exact details and other run options.

3 Storage Formats

The main data storage format used is the numpy record array, which packs structured data on disk in an efficient format. These arrays have types called dtypes, and can be queried by names inside their dtype as well as indexed names. For example, say you had the dtype of an array of 3d points in space:

```
map3d_dtype = {
  'names': ['lat', 'lon', 'alt'],
  'formats': ['float64', 'float64', 'float64'],
}
```

This means that each row of the record array will have three 64 bit floats. If you wanted to access the latitude third item in the array you would say:

```
array[3]['lat'] or array['lat'][3]
```

(see numpy record array documentation for advanced operations).

If general, you want to avoid operating on record arrays using python constructions like for loops. This is very slow. Instead, try to find a numpy operation that captures what you are trying to do (select a subset, reshape the array, etc).

3.1 Numpy record array clients

3.1.1 Database cells

Each database "cell" is a giant record array (on the order of a gigabyte for several million features SIFT, each 1024 bits long). Each "row" has the dtype

```
sift_dtype = {
  'names': ['vec', 'geom', 'index'],
  'formats': ['128uint8', '4float32', 'uint16'],
}
```

The 'vec' segment is the 1024 bit SIFT feature, and the 'geom' segment is the x, y, rotation, and hessian of feature.

Notice that the file from which the vector was extracted is not present! This is because storing filenames for each vector would be highly inefficient. Instead, they have a 16-bit index value, which indexes into another vector called the mapping. This is a map from indices->filenames.

3.1.2 Pixel to 3dspace maps

The new way: The preferred way of associating features with 3d locations is extract them from the image data directly. This was not possible with the Berkeley dataset we were using before because Earthmine did not give us the data. However with the Oakland dataset we know the exact depth and hence 3d coordinate of each pixel in the database. See reader.load_hsv_for_cell() and the calls in query.py for how this is supposed to work. If you have 3d or other data associated with your database images it is recommended that you implement in a similar way as load_hsv_for_cell(), since this cleaner strategy allows the data to be present all the way through the image pipeline.

The old way: [pixels.py] proves a way to find the 3d location of a feature (as fetched from earthmine). While it caches lookups on disk, it does not provide high throughput when accessing random features in a cell, since it has to load a file for each different image the feature is from.

To get faster access to this data (for example, if you want to use the ratio test, but only test distance scores against feature scores spatially distant: see spatial-ratio-test in query.py), use the load_3dmap_for_cell method in the reader class. This returns a packed vector where the index of the feature in the cell vector corresponds directly to the index of the 3d point in this vector.

See the file for more specific usage instructions.

3.2 The kdtree indices

Another class of large files you will (maybe) encounter are the kdtree indexes built by FLANN. Be aware that if you want to conduct ANN searches outside of [query.py], you will have to manually save these indexes for fast searches.

4 Working with features

4.1 [reader.py]

The file: [reader.py] Provides fast read/write access to the SIFT features. The sift features are extracted using some binary available on the Internet with the default settings. You must do this beforehand, which involves converting the image files to the appropriate size and format. Once you have extracted the SIFT features into the txt files,

```
get_reader(typehint)
```

returns the correct reader class for the descriptors you want to use. These reader objects abstract away caching and packing of the extracted individual sift files into a numpy array. The first time you build a database cell using

```
reader.load_cell(dir)
```

, you should get message from the reader "Reading features..." which will take a while.

4.1.1 FeatureReader

This class is returned by get_reader(), and has most of the methods for accessing database files. See [query.py] for how the reader is used. There are reader files for other descriptors besides sift. You can try these out by setting

```
C.params['descriptor']
```

to 'chog' or 'surf', but these probably won't work out of the box. The performance with other descriptors is typically much lower, so the code paths are less maintained.

4.1.2 PointToViewsMap

This class provides efficient lookups from lat/lon to earthmine views. It is also used for tag occlusion detection, and is slightly more reliable that OcclusionSummary, though not completely accurate either. This might be because we only have 3d points for feature points, not all the points in the images.

4.2 Intermediate result representations [query.py]

4.2.1 Basic Results

When you run the query system, it can produce a variety of different files as output. he most common is the .res file, which are put into the C.matchdir directory. The directory names are of the form "matchescells(g=100,r=d=236.6),queryX,{params}", so that each unique query will produce a distinct directory. This enables future runs with the same parameter to reuse the same cached files, saving lots of time.

4.2.2 Detailed Results

The query will also output a a .res-detailed.npy file in C.matchdir, which contains the feature matches as well as the count of feature votes. These are saved in a numpy packed file, which can be loaded like so:

```
import numpy
numpy.load("filename.npy")
```

The format of these files is, in python syntax:

```
[[imagename1, [{'db': (x,y), 'query': (x2,y2)}, {'db'...}, {'db'...}, ...]]
 [imagename2, [{'db': (x,y), 'query': (x2,y2)}, {'db'...}, {'db'...}, ...]]
 [imagename3, [{'db': (x,y), 'query': (x2,y2)}, {'db'...}, {'db'...}, ...]]
 ...]
```

The x,y coordinates specify the points where the features were matched. This file provides a lot more useful information that can be used in the geometric correspondence and tag transfer stages.

4.3 Feature Correspondence Computation [corr.py]

[corr.py] has a bunch of random functions related to computing features matches and a first try at pose computation. I would not recommend working with these directly. Instead, use querySystem.py as an interface. You can use the C.selection parameter to operate on a single image or subset of images in a queryset.

Important functions:

- rematch(C, Q, dbsift) runs a linear NN search on Q and the db image. You can depend on this to be fairly fast, though not fast enough for hundreds of queries.

- getSpatiallyOrdered(...) imposes a 1 dimensional spatial ordering constraint on matches by finding the longest increasing subsequence via standard dynamic programming techniques. This hasn't given much performance improvement on top of other techniques so it's not used. I believe there is prior work about spatial constraints based on rotational ordering might work better (but is significantly more complex).
- find_corr(matches, hom=False, ...) handles all homography and fundamental matrix calculations. It can filter matches by rotation and attempts multiple parameters until a good relation is found.
- isHomographyGood(H) is a fairly reliable measure of if the computed homography is sane (not upside down, scaled badly, etc).
- draw_matches(C, Q, matches, rsc_matches, ...)
- The CameraModel and compute_pose is a try at greedy search to minimize reprojection error. It currently works, but is not very good at optimizing and only minimizes lat/lon error.

5 Tags [tags.py]

[tags.py] contains several classes related to tags

- Tag holds all data about a tag and many nice utilities. For example, tag.isVisible(source) is one of the tag occlusion detection methods.
- TagCollection initializes a set of tags from a tags.csv and bearings.csv file. It matches up data from both files so that bearings.csv (which contains the normal vector of tags to their plane (tagged manually)) can be maintained seperately from the earthmine-viewer compatible tags.csv or left out altogether.
- OcclusionSummary holds earthmine information about a point. It can be queried to check what views can see a point. Unfortunately the occlusion data from earthmine is not reliable or available in all cases.

5.1 Drawing Tags [render_tags.py]

5.1.1 ImageInfo

ImageInfo is intended to be a class describing a database image. There are some functions that required you to create this class. It unifies various db image sources such as cell phones, earthmine truck, etc. Typically you would construct it in one of these ways:

```
source = render_tags.EarthmineImageInfo(dbimgpath, info)
source = render_tags.ComputedImageInfo(Q.jpgpath, Q.query_lat, Q.query_lon)
source = render_tags.AndroidImageInfo(android_image_name)
source = render_tags.get_image_info(db_img) # chooses between EarthmineImageInfo and NikonImageInfo
```

5.1.2 draw_matches()

[corr.py] currently provides tag drawing in a really messy function (which you probably want to look at for reference, but not modify directly):

```
def draw_matches(C, Q, matches, rsc_matches, H,
  inliers, db_img, out_img, matchsiftpath,
  showLine=False, showtag=True, showHom=True)
```

5.1.3 TaggedImage (WARNING: not same as TaggedImage in android.py, which is more like an ImageInfo)e

The TaggedImage class provides ways for tags to be projected onto images. The basic tag drawing methods are contained here, as are all the occlusion detection / tag culling algorithms.

To transfer tags onto an image, do the following. (NOTE that corr.py does these already!)

1. Get the source of tags. By default, context.tags will is a TagCollection of berkeley tags with normal bearings.
2. Call TaggedImage.map_tags_<method>, where tagged image is a database image. This will return tags associated with their xy coords in the database image. There are a few methods of transferring tags with different benefits/drawbacks documented in [render_tags.py] file:

   ```
   map_tags_camera(self, elat=0, elon=0, ep=0, ey=0, er=0)
   map_tags_ocs(self, C)
   map_tags_lookup(self, C)
   map_tags_hybrid
   map_tags_hybrid2
   map_tags_hybrid3
   ```

3. Use a method of projecting tags onto your image. For example, multiply each point by a homography matrix, or send the points to the mobile device to be draw, etc. Again, see draw_matches() in [corr.py] for examples.

6 Summary of Source Files

[android.py] Provides a way to read metadata from the ImageoTag Android app, which saves sensor data with each picture taken.

[config.py] Global configuration and IO utilities.

[corr.py] RANSAC, homography, and pose computation.

[earthMine.py] Earthmine API calls.

[geom.py] Geometric utilities dealing with camera projection, angle computation, distance calcuation, and image coordinate systems.

[homographyDecomposition.py] Experimental homography decomposition methods. May not work.

[info.py] Utilities for dealing with EarthMine views.

[query.py] Parallel FLANN search and voting.

[querySystem.py] The file to run.

[querySystemCopy.py] A copy of querySystem with different parameters.

[pixels.py] Gives the 3d points corresponding to 2d features in EarthMine databases.

[pnp.py] Experimental solving of the Perspective-N-Point problem, enabled by C.solve_pnp. Partially implemented.

[posit.py] Experimental POSIT algorithm usage, enabled by C.do_posit. Does not work very well.

[reader.py] Database IO methods.

[render_tags.py] Tag rendering/culling code.

[system.py] The “black boxes” of the system.

[tags.py] Tag data structures.

[tags.csv] The tags in EarthMine Viewer format.

[tags-canonical.csv] Tag data including normal vectors. Intended to supplement tags.csv.

[util.py] Generic utility methods for cell computation and file access.